Future Generation Computer Systems 78 (2018) 77–86

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A novel binary artificial bee colony algorithm for the set-union knapsack problem

Yichao He [a], Haoran Xie [b], Tak-Lam Wong [b], Xizhao Wang [c,*]

[a] College of Information and Engineering, Hebei GEO University, Shijiazhuang 050031, China
[b] Department of Mathematics and Information Technology, The Education University of Hong Kong, Hong Kong Special Administrative Region
[c] College of Computer Science and Software Engineering, Shenzhen University, Shenzhen 518060, China

HIGHLIGHTS

- A novel bee colony method based on the full mapping function is proposed.
- Infeasible solutions are addressed by using a greedy strategy for Knapsack problems.
- The method has better results than extant approximation algorithms to solve SUKP.
- The proposed generic model can be integrated with other evolutionary algorithms.

ARTICLE INFO

Article history:
Received 3 September 2016
Received in revised form 19 April 2017
Accepted 19 May 2017
Available online 15 June 2017

Keywords:
Set-union knapsack problem
Artificial bee colony
Infeasible solution
Repairing and optimization

ABSTRACT

This article investigates how to employ artificial bee colony algorithm to solve Set-Union Knapsack Problem (SUKP). A mathematical model of SUKP, which is to be easily solved by evolutionary algorithms, is developed. A novel binary artificial bee colony algorithm (BABC) is also proposed by adopting a mapping function. Furthermore, a greedy repairing and optimization algorithm (S-GROA) for handling infeasible solutions by employing evolutionary technique to solve SUKP is proposed. The consolidation of S-GROA and BABC brings about a new approach to solving SUKP. Extensive experiments are conducted upon benchmark datasets for evaluating the performance of our proposed models. The results verify that the proposed approach is significantly superior to the baseline evolutionary algorithms for solving SUKP such as A-SUKP, ABC_{bin} and binDE in terms of both time complexity and solution performance.

1. Introduction and background

The Set-Union Knapsack Problem (SUKP) [1,2], a natural extension of the standard 0–1 Knapsack Problem (0–1 KP), is an NP-complete problem. In spite of the difficulty, SUKP has been identified to be valuable in various domain-specific applications such as financial decision making [2,3], flexible manufacturing machine [1,4,5], database partitioning [6,7], smart city [8] and data stream compression [9]. In particular, a popular application of SUKP is to build public key prototype (PKC) [10]. To enhance the security in building PKC based on SUKP, researchers attempt to hidden the trace of public key through many iterations. Therefore, intruders are unable to adopt Lenstra integer programming algorithms to break the key. It is worth to pointing out that Evolutionary Algorithms (EAs) are essentially a random search scheme in which the search performance is irrelevant to properties of the problem. Many researchers therefore believe that EA-based PKC is a promising technique. The extant studies show that the crucial techniques are how to design quick and efficient algorithm to solve SUKPs. It is commonly acknowledged that studies about approaches to solving SUKP based on EAs are quite important to the area of information security.

Goldschmidt et al. proposed a Dynamic Programming (DP) algorithm for addressing SUKP accurately based on the hypergraph theory [1]. However, the time complexity of the algorithm is exponential so that it is infeasible for real-world applications. Moreover, Ashwin developed an approximation algorithm called A-SUKP to solve SUKP based on the greedy strategy [2]. The approximation rate of A-SUKP is $1/(1-e^{-1/d})$, where $d(d \geq 2)$ is the upper-bound of the occurrence of all the elements. Apparently, the approximation solution of A-SUKP is relatively unsatisfactory and inefficient when d becomes large.

Artificial Bee Colony (ABC), which is a swarm intelligence approach developed in 2005 [11–13], mimics the bee colony to search for quality honey in the natural environment. Karaboga and Basturk analyzed the characteristics and conducted comprehensive

* Corresponding author.
E-mail address: xzwang@szu.edu.cn (X. Wang).

http://dx.doi.org/10.1016/j.future.2017.05.044
0167-739X/

comparisons between ABC and a number of Evolutionary Algorithms (EAs) including Genetic Algorithm (GA) [14], Differential Evolution (DE) [15], Particle Swarm Optimization (PSO) [16], etc. The experimental results illustrate that the ABC has better than or similar performance to other population-based algorithms. In addition, ABC has the advantage of employing a few tuning parameters. Recently, ABC has achieved promising results in several optimization problems. For example, Karaboga proposed a method for digital infinite impulse response (IIR) filters based on ABC [17]. Karaboga and Ozturk exploited ABC for training network networks [18]. Kang et al. proposed an algorithm called Rosenbrock Artificial Been Colony algorithm to improve the accuracy of ABC in numerical optimization problems [19]. Karaboga and Ozturk proposed a novel clustering approach based on ABC [20]. Pan et al. developed a Discrete Artificial Bee Colony algorithm to address the lot-streaming flow shop scheduling problem [21]. Tsai adapted ABC for tackling the constrained optimization problems [22]. Kiran et al. developed a variable search strategy for continuous optimization problems [23]. Banitalebi et al. presented an approach called Enhanced Compact Artificial Bee Colony, which improves the performance of ABC [24]. Kiran proposed a new approach for the incapacitated facility location problem based on Continuous Artificial Bee Colony (ABC_{bin}) algorithm [25]. Ozturk et al. proposed a Binary Artificial Bee Colony (BABC) algorithm for solving the 0–1 KP problem by adopting genetic operators [26]. Secui designed an enhanced ABC for solving the economic dispatch problem [27]. Due to the large number of applications of ABC to optimization problems, this article aims at developing an approach to solving SUKP based on ABC. We firstly introduce a mathematical model of SUKP. One characteristic of the model is that EAs can also be readily integrated to solve SUKP. A novel Binary Artificial Bee Colony (BABC) algorithm is proposed. Furthermore, we present a novel method for solving SUKP by an efficient algorithm (named S-GROA) to tackle the issue of the infeasible solution in SUKP.

The remaining parts of the paper is organized as follows. Section 2 presents the definition of SUKP and the mathematical model of SUKP, which facilitate the use of EAs. Section 3 reviews the principle of ABC and proposes BABC algorithm. In Section 4, we firstly present an efficient and generic an efficient algorithm (named S-GROA) to tackle the issue of the infeasible solution in SUKP. In addition, we applied S-GROA to BABC to accomplish a new approach for solving SUKP. It follows by a large-scale comparison between BABC and other methods including A-SUKP, GA, ABC_{bin} and binDE [28] for solving three real-world problems in Section 5. The experimental results show that BABC is superior to A-SUKP with respect to efficiency and performance. Moreover, the performance of BABC is more efficient than one of GA, ABC_{bin} and binDE. We draw the conclusion and discuss the future direction of research in Section 6.

2. Definition and mathematical models

Definition 1 (Set-union Knapsack Problem, SUKP) [1,2]: Given a set of elements $U = \{1, 2, \ldots, n\}$ and a set of items $S = \{1, 2, \ldots, m\}$, such that each item $i \in S$ $(i = 1, 2, \ldots, m)$ corresponds to a subset $U_i \subseteq U$ and associates with a value $p_i > 0$. Each element $j \in U$ $(j = 1, 2, \ldots, n)$ has a weight $w_j > 0$. For an arbitrary non-empty set $A \subseteq S$, the profit and the weight of A are defined as $P(A) = \sum_{i \in A} p_i$ and $W(A) = \sum_{j \in \bigcup_{i \in A} U_i} w_j$ respectively. The objective of SUKP is to find the subset $S^* \subseteq S$, such that $P(S^*)$ is maximized, subject to $W(S^*) \leq C$ where C is the capacity of knapsack.

Formally, SUKP can be notated as follows:

$$\textit{Maximize} \quad P(A) = \sum_{i \in A} p_i \tag{1}$$

$$\text{subject to} \quad W(A) = \sum_{j \in \bigcup_{i \in A} U_i} w_j \leq C, A \subseteq S. \tag{2}$$

We name the above mathematical model as SUKP-M1. Without loss of generality, let p_i $(i = 1, 2, \ldots, m)$, w_j $(j = 1, 2, \ldots, n)$ and C be positive integers. Let $\boldsymbol{V} = \{U_1, U_2, \ldots, U_m\}$ be the cover of U, such that $U_i \subset U (i = 1, 2, \ldots, m)$ and $U_i \neq \Phi$. In addition, $W(S) > C$ and $\sum_{j \in U_i} w_j \leq C$ for all $i \in S$. EAs are not suitable to use SUKP-M1. Therefore, we developed an integer programming model named as SUKP-M2, which can facilitate for solving SUKP by using EAs.

Let $Y = [y_1, y_2, \ldots, y_m] \in \{0, 1\}^m$ be an m-dimension 0–1 vector. $A_Y = \{i | y_i \in Y, y_i = 1, 1 \leq i \leq m\} \subseteq S$, then for an arbitrary $i = 1, 2, \ldots, m$, $y_i = 1$ if and only if $i \in A_Y$. Apparently, the 0–1 vector Y and the subset $A_Y \subset S$ are one-to-one mappings. By using the one-to-one mapping of Y and A_Y, SUKP can be modeled as a new integer programming model (SUKP-M2) as follows.

$$\textit{Maximize} \quad f(Y) = \sum\nolimits_{i=1}^{m} y_i p_i \tag{3}$$

$$\text{subject to} \quad W(A_Y) = \sum\nolimits_{j \in \bigcup_{i \in A_Y} U_i} w_j \leq C. \tag{4}$$

According to SUKP-M2, all 0–1 vectors $Y = [y_0, y_1, \ldots, y_m] \in \{0, 1\}^m$ are the only possible solutions of SUKP. Solutions which satisfy constraint (4) are feasible solutions of SUKP, while the solutions which cannot satisfy constraint (4) are infeasible solutions. As an instance of SUKP is consisted of the parameters n and m, the profit set $P = \{p_i | 1 \leq i \leq m\}$, the weight set $W = \{w_j | 1 \leq j \leq n\}$, the subset family $\boldsymbol{V} = \{U_1, U_2, \ldots, U_m\}$ and the capacity C, an instance of SUKP will be denoted as **INS**$(n, m, P, W, C, \boldsymbol{V})$ in the remaining sections.

3. Binary ABC (BABC) algorithm

3.1. Artificial bee colony algorithm

The Artificial Bee Colony (*ABC*) [11,12] is a population-based meta-heuristic algorithm for optimizing numerical problems. It was inspired by the intelligent foraging behavior of honey bees when they are seeking a quality food source. In the *ABC*, each candidate solution to the optimization problem is associated with a food source. It is represented by an D-dimensional real-coded vector, where D is the dimension of the optimization problem. The quality (i.e., fitness) of a solution corresponds to the amount of nectar in that food source.

The population of *ABC* includes three categories of bees: employed bees, onlooker bees and scout bees. Each of them carrying on various activities to find a better food source. Employed bees take charge of exploring the solution space to search for food sources and then sharing various pieces of information with other bees. Onlooker bee exerts a probabilistically modification on the solution (food source) for finding a new solution and tests the fitness amount of the new solution. Scout bees work to help *ABC* escape from local optimums. To sum up, the employed and onlooker bees are responsible for exploitation, whereas the scout bees handle exploration.

Let $X = [x_1, x_2, \ldots, x_D] \in [l_j, u_j]^D$ represents a food source, *fit* (X) is the fitness value of a food source as shown in the following Eq. (5).

$$\textit{fit}(X) = \begin{cases} 1/(1 + f(X)), & \textit{if } f(X) \geq 0 \\ 1 + |f(X)|, & \textit{otherwise} \end{cases} \tag{5}$$

where $f(X)$ is the objective function of X, l_j and u_j represent minimum and maximum of the jth variable respectively. In the *ABC*,

each iteration of the search is consisted of three phases including an employed bee phase, an onlooker bee phase and a scout bee phase. These three phases of *ABC* are detailed as follows.

Employed bee phase: Each employed bee attempts to find a new food source in order to improve the self-solution by using Eq. (6).

$$v_{ij} = x_{ij} + \varphi_{ij} * (x_{ij} - x_{kj}), \tag{6}$$

where $i, k \in \{1, 2, \ldots, N\}$ and $i \neq k, j = 1, 2, \ldots, D$, N is the size of the population, $V_i = [v_{i1}, v_{i2}, \ldots, v_{iD}] \in [l_j, u_j]^D$ is ith candidate food source, $X_i = [x_{i1}, x_{i2}, \ldots, x_{iD}] \in [l_j, u_j]^D$ is ith employed bee, $X_k = [x_{k1}, x_{k2}, \ldots, x_{kD}] \in [l_j, u_j]^D$ is kth employed bee, φ_{ij} is a uniformly distributed random real number in range of $[-1,1]$. A greedy selection is applied between V_i and X_i by retaining the better one. In other words, V_i will be assigned to X_i ($X_i = V_i$) if *fit* (V_i) is less than *fit* (X_i), otherwise X_i remains the same.

Onlooker bee phase: The onlooker bee chooses a food source by issuing a precise search, which depends on their fitness. The probability of selection is calculated as follows:

$$p_i = fit(X_i) / \sum_{j=1}^{N} fit(X_j) \tag{7}$$

where p_i is the probability of being selected ith employed bee. The onlooker bees then try to improve the solutions of employed bee by using Eq. (6). Similar to the employed bee phase, the greedy selection is also applied to the new food source and old one.

Scout bee phase: If a food source has not been updated after several consecutive iterations (i.e., a threshold is set), the employed bee associated with that food source will become a scout bee. The scout bee then executes a random initialization in the solution space by using Eq. (8) to find a new food source. If the scout bee finds a new source, she will become an employed bee again.

$$x_{ij} = l_j + rand(0, 1) * (u_j - l_j), \tag{8}$$

where $i = 1, 2, \ldots, N, j = 1, 2, \ldots, D$. *limit* is a pre-set threshold and *rand* (0,1) is a random real number within [0,1].

3.2. Binary artificial bee colony (BABC) algorithm

In ABC, each food source is denoted by a real vector, which cannot be directly used for solving binary optimization problems. To exploit the artificial bee colony for solving binary optimization problems, the food source needs to be denoted as the candidate solution of binary optimization problems. Therefore, a surjection mapping, which can map a real vector to a 0–1 vector, is adopted to propose a novel binary artificial bee colony (BABC) algorithm for solving binary optimization problems.

If a real vector $X = [x_1, x_2, \ldots, x_D]$ for representing a food source is constrained in $[-a, a]^D$ (where a is a positive real number), there are two cases $x_j \geqslant 0$ and $x_j < 0 (j = 1, 2, \ldots, D)$ for each component vector x_j in X. Let $y_j = 1$ when $x_j \geqslant 0$ and $y_j = 0$ when $x_j < 0$. According to the sign (negative or positive) of the component vector in each dimension in the real vector X, a 0–1 vector $Y = [y_1, y_2, \ldots, y_D] \in \{0, 1\}^D$ can be obtained. Accordingly, we can define a surjection mapping $\Psi : [-a, a]^D \to \{0, 1\}^D$, where $Y = \Psi(X)$ and

$$y_j = \begin{cases} 1, & if\ x_j \geq 0 \\ 0, & if\ x_j < 0 \end{cases} \tag{9}$$

where $j = 1, 2, \ldots, D$. By employing the mapping Ψ, each real vector can be mapped into a 0–1 vector in $\{0, 1\}^D$.

In *BABC*, we re-define the fitness of food sources by introducing the mapping Ψ. For each real vector of food sources $X \in [-a, a]^D$, we generate a 0–1 vector $Y \in \{0, 1\}^D$ by using mapping Ψ. Considering Y as potential solutions of the corresponding binary optimization problem, we use $f(Y)$ as the fitness *fit* (X) of food sources. For example, $f(X)$ is replaced by $f(Y)$ in Eq. (5) to calculate *fit* (X) (or let *fit* (X) $= f(Y)$). Therefore, we propose a novel binary artificial bee colony (BABC) algorithm for solving binary optimization problems by using above mechanism.

Let $Y_B = [y_{B1}, y_{B2}, \ldots, y_{BD}] \in \{0, 1\}^D$ be a feasible solution of the optimal food source in BABC, and *MIT* be the number of iterations of BABC. The detail steps of the *BABC* algorithm are shown in Algorithm 1.

Algorithm 1. ***BABC***

Input: Instance of binary optimization problem; parameters a, N, *limit* and *MIT*.

Output: Approximate solution or optimal solution Y_B, and objective function value $f(Y_B)$.

1. Initialization
   - 1.1 Use Eq.(8) to randomly generate $X_i \in [-a,a]^D$, calculate $Y_i = \Psi(X_i)$, $T_i \leftarrow 1$, i=1 to N.
   - 1.2 Use $f(Y_i)$ to calculate $fit(X_i)$, and determine Y_B, $t \leftarrow 1$.
2. Employed bee phase
   - 2.1 Use Eq.(6) to generate $V_i \in [-a,a]^D$, $Z_i = \Psi(V_i)$, and use $f(Z_i)$ to calculate $fit(V_i)$, i=1 to N.
   - 2.2 If $fit(V_i)$ is better than $fit(X_i)$, replace X_i by V_i, $T_i \leftarrow 1$; otherwise $T_i \leftarrow T_i + 1$.
3. Onlooker bee phase
   - 3.1 Use Eq.(7) to calculate p_i, i=1 to N.
   - 3.2 For i=1,2,…,N, if p_i>rand(0,1), execute step 3.3 ~ 3.4
   - 3.3 Use Eq.(6) to generate $V_i \in [-a,a]^D$, $Z_i = \Psi(V_i)$, and use $f(Z_i)$ to calculate $fit(V_i)$.
   - 3.4 If $fit(V_i)$ is better than $fit(X_i)$, replace X_i by V_i, $T_i \leftarrow 1$; otherwise $T_i \leftarrow T_i + 1$.
4. Scout bee phase
   - 4.1 For i=1,2,…,N, if T_i>*limit*, execute step 4.2.
   - 4.2 Use Eq.(8) to generate $X_i \in [-a,a]^D$, $Y_i = \Psi(X_i)$, use calculate $f(Y_i)$ to $fit(X_i)$, $T_i \leftarrow 1$.
5. Termination criteria
   - 5.1 Determine Y_B, $t \leftarrow t + 1$.
   - 5.2 If t>*MIT*, output Y_B and $f(Y_B)$ and terminate the algorithm; otherwise, go to step 2.

In Algorithm 1, the time complexity of Step 1 to Step 4 is $O(D * N)$. Therefore, the overall time complexity of *BABC* is $O(MIT * D * N)$.

4. Apply BABC to solve SUKP

4.1. Greedy repairing and optimization

As SUKP is a constrained optimization problem, it is inevitable to generate infeasible solutions when BABC is used to solve SUKP. The existence of infeasible solutions not only reduces the effective of the algorithm but also results in difficulty of employing the objective function of solutions to calculate the fitness of food sources. Therefore, the key problem is how to handle infeasible solutions generated by the algorithm.

It is well known that the common approaches to handling infeasible solutions are the penalty function methods and repair algorithms [29,30]. The penalty function method adopts an appropriate penalty term to appropriately "penalty" the objective function value of infeasible solution, which gives a reasonable measurement for a infeasible solution. However, the infeasible solutions can neither be eliminated nor improve the quality, so that EAs do not have good performance for solving combinatorial optimization problems. Michalewicz [31] compared the penalty function methods and repair algorithms by using GA to solve 0–1 knapsack problems, and then found that repair algorithms are more suitable for handling the infeasible solutions of 0–1 KP. He et al. [32–34] added the optimization techniques to improve the original repair algorithm to an repairing and optimization algorithm when they use the GA to solve 0–1 KP, the randomized time-varying knapsack problems and the discounted {0–1} knapsack problems. The repairing and optimization algorithm can not only repair infeasible solutions to feasible ones but also improve the quality of solutions. Based on the above idea and a greedy strategy [2], a greedy repairing and optimization algorithm (named S-GROA) is proposed to handle infeasible solutions.

According to the greedy strategy [2], let d_j ($j = 1, 2, \ldots, n$) be frequency of the element $j (j \in U)$ in the subsets $U_1, U_2, \ldots, U_m$,

$R_i = \sum_{j\in U_i}(w_j/d_j)\,(i = 1, 2, \ldots, m)$. The algorithm *QuickSort* [35] is employed to sort all items in S in descending order according to the metric p_i/R_i $(i = 1, 2, \ldots, m)$, and the index of each item are then stored in a one-dimensional array H[1 . . . m] according to sorted order. For any m-dimensional 0–1 vector $Y = [y_1, y_2, \ldots, y_m] \in \{0, 1\}^m$, we simplify the notation as $A_Y = \{i | y_i \in Y$ and $y_i = 1, 1 \le i \le m\}$. The detail steps of *S-GROA* are shown in Algorithm 2.

Algorithm 2. ***S-GROA***

Input: SUKP potential solutions $Y=[y_1, y_2,\ldots,y_m]\in\{0,1\}^m$ and array H[1…m].

Output: The feasible solution $Y=[y_1, y_2,\ldots,y_m]\in\{0,1\}^m$ and objective function value $f(Y)$.

1 **if** ($W(A_Y)\le C$) **then goto** 7
2 $Z\leftarrow[0,0,\ldots,0]$. % Note: $Z=[z_1,z_2,\ldots,z_m]$ is a m-dimensional 0-1 vector %
3 **for** $i\leftarrow 1$ **to** m **do** %repairing stage%
4 **if** ($y_{H[i]}$=1 and $W(A_Z\cup\{H[i]\})\le C$) **then** $z_{H[i]}\leftarrow 1$ and $A_Z\leftarrow A_Z\cup\{H[i]\}$.
5 **end for**
6 $Y\leftarrow Z$.
7 **for** $i\leftarrow 1$ **to** m **do** %optimizing stage%
8 **if** ($y_{H[i]}$=0 and $W(A_Y\cup\{H[i]\})\le C$) **then** $y_{H[i]}\leftarrow 1$ and $A_Y\leftarrow A_Y\cup\{H[i]\}$.
9 **end for**
10 **return**($Y, f(Y)$).

In *S-GROA*, step 1 determines whether the potential solution Y is a feasible solution or not. If it is a feasible solution, the algorithm will move to the optimizing stage (step 7 to 9). Otherwise, a repairing stage (step 3 to 6) will be firstly executed to obtain a feasible solution, and then do the optimization stage. In step 10, $f(Y) = \sum_{i=1}^{m} y_i p_i$ is the output of objective function of feasible solution Y. The time complexity of *S-GROA* is O(nm), which is an efficient method for handling the infeasible solutions of SUKP.

4.2. Application of BABC to SUKP

To exploit *S-GROA* for addressing infeasible solutions and improving their quality when the *BABC* is employed to address SUKP, Algorithm 1 needs to be further modified in following three aspects:

(1) Before the generation of the initial populations, all items in $S$are sorted in descending order according to the metric p_i/R_i $(i = 1, 2, \ldots, m)$, and the index of each item are then stored in a one-dimensional array H[1 . . . m] according to sorted order.

(2) For all feasible solutions and potential solutions respect to food sources in the initial population, *S-GROA* is employed for repairing and optimizing. The output of objective function of feasible solution is considered as the fitness.

(3) For the feasible and potential solutions of food sources generated in a iteration, *S-GROA* is firstly employed for repairing and optimizing them. The output of objective function of feasible solutions are then considered as the fitness of generated food sources.

To simplify the notations in the algorithm for *BABC* to solve SUKP, the following notation is given as

$$\mathrm{H}[1\ldots m] \leftarrow QuickSort(p_i/R_i, \mathbf{INS}(n, m, P, W, C, \boldsymbol{V})) \quad (10)$$

where *QuickSort* [35] is used for sorting all items in instance $\mathbf{INS}(n, m, P, W, C, \boldsymbol{V})$ to descending order according to the metric p_i/R_i $(i = 1, 2, \ldots, m)$, and all item's index are then stored in an array H[1 . . . m]. The detail steps of *BABC* for *SUKP* are shown in Algorithm 3.

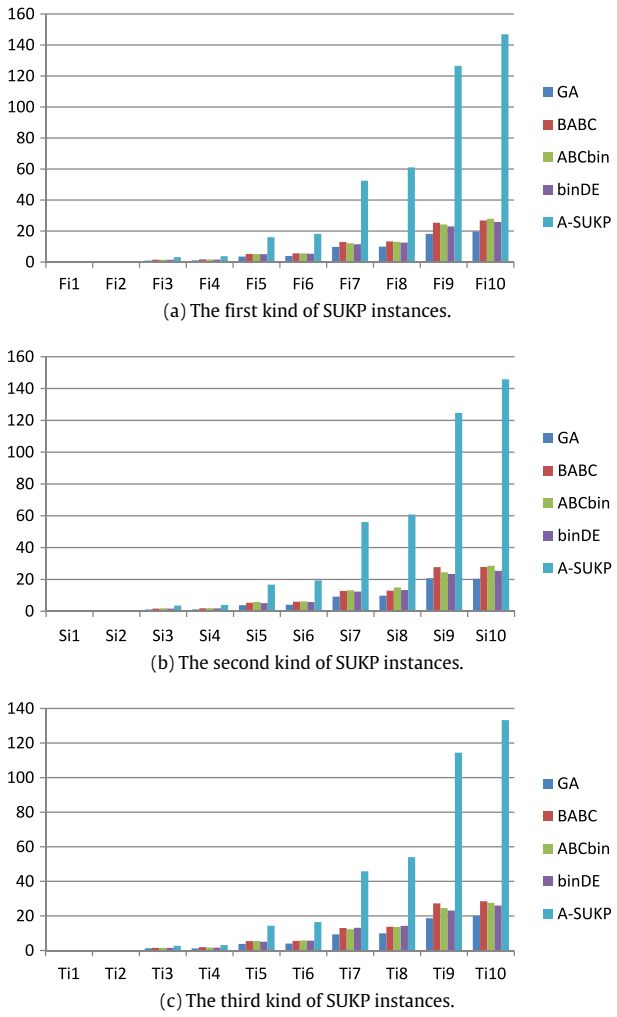


Fig. 1. The time cost of *GA*, *BABC*, ABC_{bin}, *binDE* and *A-SUKP* for solving three kinds of SUKP instances.

Algorithm 3. ***BABC for SUKP***

Input: **INS**(n, m, P, W, C, $\boldsymbol{V}$); Parameter a, N, *limit*, and *MIT*.

Output: approximate solution or optimized solution Y_B and the objective function value $f(Y_B)$.

1. H[1…m]←*QuickSort*(p_i/R_i, **INS**(n, m, P, W, C, $\boldsymbol{V}$)).
2. Initialization
   2.1 Use Eq.(8) to randomly generate $X_i\in[-a,a]^m$, calculate $Y_i=\Psi(X_i)$, $T_i\leftarrow 1$, i=1,2,…,N.
   2.2 $(Y_i, f(Y_i))\leftarrow$***S-GROA***(Y_i, H[1…m]). %***S-GROA*** is shown in Algorithm 2%
   2.3 Use $f(Y_i)$ to calculate $fit(X_i)$, determine Y_B, $t\leftarrow 1$.
3. Employed bee phase
   3.1 Use Eq.(6) to produce $V_i\in[-a,a]^m$, calculate $Z_i=\Psi(V_i)$, i=1,2,…,N.
   3.2 $(Z_i, f(Z_i))\leftarrow$***S-GROA***(Z_i, H[1…m]) , and use $f(Z_i)$ to calculate $fit(V_i)$.
   3.3 If $fit(V_i)$ is better than $fit(X_i)$, replace X by V_i, $T_i\leftarrow 1$; otherwise $T_i\leftarrow T_i+1$.
4. Onlooker bee phase
   4.1 Use Eq.(7) to calculate p_i, i=1 to N.
   4.2 For i=1,2,…,N, if p_i>rand(0,1), execute step 4.3~ 4.5
   4.3 Use Eq.(6) to generate $V_i\in[-a,a]^m$, calculate $Z_i=\Psi(V_i)$.
   4.4 $(Z_i, f(Z_i))\leftarrow$***S-GROA***(Z_i, H[1…m]), and use $f(Z_i)$ to calculate $fit(V_i)$.
   4.5 If $fit(V_i)$ is better than $fit(X_i)$, replace X_i by V_i,$T_i\leftarrow 1$; otherwise $T_i\leftarrow T_i+1$.
5. Scout bee phase
   5.1 For i=1,2,…,N, if T_i>*limit*, execute step 5.2~5.3.
   5.2 Use Eq.(8) to generate $X_i\in[-a,a]^m$, calculate $Y_i=\Psi(X_i)$.
   5.3 $(Y_i, f(Y_i))\leftarrow$***S-GROA***(Y_i, H[1…m]), and calculate $fit(X_i)$ by $f(Y_i)$, $T_i\leftarrow 1$.
6. Termination criteria
   6.1 Determine Y_B, $t\leftarrow t+1$.
   6.2 If t>*MIT*, output Y_B and $f(Y_B)$ and terminate the algorithm; otherwise, go to step 2.

Table 1
The numbering of all SUKP instances.

ID	The first class ($m > n$)	ID	The second class ($m = n$)	ID	The third class ($m < n$)
Fi1	sukp 100_85_0.1_0.75	Si1	sukp 100_100_0.1_0.75	Ti1	sukp 85_100_0.1_0.75
Fi2	sukp 100_85_0.15_0.85	Si2	sukp 100_100_0.15_0.85	Ti2	sukp 85_100_0.15_0.85
Fi3	sukp 200_185_0.1_0.75	Si3	sukp 200_200_0.1_0.75	Ti3	sukp 185_200_0.1_0.75
Fi4	sukp 200_185_0.15_0.85	Si4	sukp 200_200_0.15_0.85	Ti4	sukp 185_200_0.15_0.85
Fi5	sukp 300_285_0.1_0.75	Si5	sukp 300_300_0.1_0.75	Ti5	sukp 285_300_0.1_0.75
Fi6	sukp 300_285_0.15_0.85	Si6	sukp 300_300_0.15_0.85	Ti6	sukp 285_300_0.15_0.85
Fi7	sukp 400_385_0.1_0.75	Si7	sukp 400_400_0.1_0.75	Ti7	sukp 385_400_0.1_0.75
Fi8	sukp 400_385_0.15_0.85	Si8	sukp 400_400_0.15_0.85	Ti8	sukp 385_400_0.15_0.85
Fi9	sukp 500_485_0.1_0.75	Si9	sukp 500_500_0.1_0.75	Ti9	sukp 485_500_0.1_0.75
Fi10	sukp 500_485_0.15_0.85	Si10	sukp 500_500_0.15_0.85	Ti10	sukp 485_500_0.15_0.85

Table 2
The computing results of the first kind of SUKP instances.

Instance	Results	*A-SUKP*	*GA*	*BABC*	ABC_{bin}	*binDE*
sukp 100_85_0.10_0.75	*Best*	12 459	13 044	**13 251**	13 044	130 44
	Mean	12 459	12 956.4	**13 028.5**	12 818.5	12 991
	StD	0.00	130.66	92.63	153.06	75.95
	Time	0.218	0.112	0.210	0.202	0.196
sukp 100_85_0.15_0.85	*Best*	11 119	12 066	**12 238**	**12 238**	12 274
	Mean	11 119	11 546	**12 155**	12 049.3	12 123.9
	StD	0.00	214.94	53.29	96.11	67.61
	Time	0.235	0.119	0.223	0.223	0.217
sukp 200_185_0.10_0.75	*Best*	11 292	13 064	**13 241**	12 946	**13 241**
	Mean	11 292	12 492.5	**13 064.4**	11 861.5	12 940.7
	StD	0.00	320.03	99.57	324.65	205.70
	Time	3.156	1.013	1.562	1.534	1.479
sukp 200_185_0.15_0.85	*Best*	12 262	13 671	**13 829**	13 671	13 671
	Mean	12 262	12 802.9	**13 359.2**	12 537	13 110
	StD	0.00	291.66	234.99	289.53	269.69
	Time	3.766	1.133	1.729	1.699	1.651
sukp 300_285_0.10_0.75	*Best*	8941	**10 553**	10 428	9751	10 420
	Mean	8941	9980.87	**9994.76**	9339.3	9899.24
	StD	0.00	142.97	154.03	158.15	153.18
	Time	16.079	3.608	5.281	5.144	5.109
sukp 300_285_0.15_0.85	*Best*	9432	11 016	**12 012**	10 913	11 661
	Mean	9432	10 349.8	**10 902.9**	9 957.85	10 499.4
	StD	0.00	215.13	449.45	276.90	403.95
	Time	18.204	3.899	5.673	5.567	5.355
sukp 400_385_0.10_0.75	*Best*	9076	10 083	**10 766**	9674	10 576
	Mean	9076	9641.85	**10 065.2**	9187.76	9681.46
	StD	0.00	168.94	241.45	167.08	275.05
	Time	52.522	9.779	12.976	12.161	11.485
sukp 400_385_0.15_0.85	*Best*	8514	**9831**	9649	8978	9649
	Mean	8514	**9326.77**	9135.98	8539.95	9020.87
	StD	0.00	192.20	151.90	161.83	150.99
	Time	61.035	9.978	13.359	13.077	12.548
sukp 500_485_0.10_0.75	*Best*	9864	**11 031**	10 784	10 340	10 586
	Mean	9864	**10 567.9**	10 452.2	9910.32	10 363.8
	StD	0.00	123.15	114.35	120.82	93.39
	Time	126.569	18.198	25.372	24.251	23.021
sukp 500_485_0.15_0.85	*Best*	8299	**9472**	9090	8759	9191
	Mean	8299	8692.67	**8857.89**	8365.04	8783.99
	StD	0.00	180.12	94.55	114.10	131.05
	Time	146.93	19.720	26.874	27.969	25.893

The time complexity of (i) computing p_i/T_i is O(n); (ii) *QuickSort* is O($m \log m$); and (iii) *S-GROA* is O(mn). Both *MIT* and *N* are the constant times of *Max* {m, n}, so the time complexity of Algorithm 3 is O(M^4), where M =*Max* {m, n}. In other words, it is a random approximation algorithm with the polynomial time complexity for solving SUKP.

5. Experimental results and discussions

To examine the performance of BABC for solving SUKP, *A-SUKP*, *BABC*, *GA* [14], ABC_{bin} [25] and *binDE* [28] are adopted to solve three kinds of SUKP instances, and their computing results are compared. The experimental platform is an Acer Aspire E1-570G laptop with Intel(R) Core(TM)i5-3337u CPU-1.8 GHz, 4 GB DDR3 (3.82 GB usable). The operation system is Microsoft Windows 8. All algorithms are implemented in C++ under the IDE of Visual C++ 6.0. The line charts are implemented by MATLAB7.10.0.499 (R2010a).

5.1. SUKP instance and parameter settings of the algorithm

As there is no benchmark for the SUKP, we propose a method to generate three kinds of instances of SUKP, and give a naming rule of SUKP instance and give a dataset of three kinds of SUKP instance.

To simplify calculations, an $m \times n$ 0–1 matrix $\mathbf{M} = (r_{ij})$ denotes subset family $\boldsymbol{V} = \{U_1, U_2, \ldots, U_m\}$. For each element r_{ij} in $\mathbf{M}$ ($i = 1, 2, \ldots, m$; $j = 1, 2, \ldots, n$), $r_{ij} = 1$ only if $j \in U_i$. To illustrate the value settings of parameters of SUKP instances, we name the SUKP instances as sukp $m\_n\_\alpha\_\beta$ uniformly,

Table 3
The computing results of the second kind of SUKP instances.

Instance	Results	*A-SUKP*	*GA*	*BABC*	ABC_{bin}	*binDE*
sukp 100_100_0.10_0.75	*Best*	13 634	**14 044**	13 860	13 860	13 814
	Mean	13 634	**13 806**	13 734.9	13 547.2	13 675.9
	StD	0.00	144.91	70.76	119.11	119.53
	Time	0.264	0.129	0.213	0.225	0.225
sukp 100_100_0.15_0.85	*Best*	11 325	13 145	**13 508**	13 498	13 407
	Mean	11 325	12 234.8	**13 352.4**	13 103.1	13 212.8
	StD	0.00	388.66	155.14	343.46	287.45
	Time	0.281	0.143	0.244	0.249	0.248
sukp 200_200_0.10_0.75	*Best*	10 328	11 656	**11 846**	11 191	11 535
	Mean	10 328	10 888.7	**11 194.3**	10 424.1	10 969.4
	StD	0.00	237.85	249.58	197.88	302.52
	Time	3.578	1.106	1.633	1.839	1.634
sukp 200_200_0.15_0.85	*Best*	9784	**11 792**	11 521	11 287	11 469
	Mean	9784	10 827.5	**10 945**	10 345.9	10 717.1
	StD	0.00	334.43	255.14	273.47	341.08
	Time	3.953	1.183	1.819	1.932	1.797
sukp 300_300_0.10_0.75	*Best*	10 208	12 055	12 186	11 494	**12 304**
	Mean	10 208	11 755.1	**11 945.8**	10 922.3	11 864.4
	StD	0.00	144.45	127.80	182.63	160.42
	Time	16.751	3.789	5.315	5.855	5.118
sukp 300_300_0.15_0.85	*Best*	9183	**10 666**	10 382	9633	10 382
	Mean	9183	**10 099.2**	9859.69	9186.87	9710.37
	StD	0.00	337.42	177.02	147.78	208.48
	Time	19.391	4.106	6.019	6.225	5.801
sukp 400_400_0.10_0.75	*Best*	9751	10 570	**10 626**	10 160	10 462
	Mean	9751	**10 112.4**	10 101.1	9549.04	9975.8
	StD	0.00	157.89	196.99	141.27	185.57
	Time	56.065	9.187	12.805	13.189	12.358
sukp 400_400_0.15_0.85	*Best*	8497	9235	**9541**	9033	9388
	Mean	8497	8793.76	**9032.95**	8365.62	8768.42
	StD	0.00	169.52	194.18	153.40	212.24
	Time	60.816	9.830	12.953	14.989	13.334
sukp 500_500_0.10_0.75	*Best*	9615	10 460	**10 755**	10 071	10 546
	Mean	9615	10 185.4	**10 328.5**	9738.17	10 227.7
	StD	0.00	114.19	91.615	111.63	103.32
	Time	124.678	20.717	27.735	24.450	23.468
sukp 500_500_0.15_0.85	*Best*	7883	**9496**	9318	9262	9312
	Mean	7883	8882.88	**9180.74**	8617.91	9096.13
	StD	0.00	158.21	84.91	141.32	145.45
	Time	145.789	20.379	27.813	28.632	25.417

where m is the number of items in instances, n is the number of elements, α denotes the density of element 1 in the matrix $\mathbf{M}$, i.e., $\alpha = \left(\sum_{i=1}^{m}\sum_{j=1}^{n} r_{ij}\right)/(nm)$, and β is the ratio of C to the sum of all elements, i.e., $\beta = C/\sum_{j=1}^{n} w_j$. According to the relationship between m and n, there are three kinds of SUKP instances: (1) 10 SUKP instance with $m > n$, (2) 10 SUKP instance with $m = n$, and (3) 10 SUKP instance with $m < n$. The datasets can be found in http://sncet.com/ThreekindsofSUKPinstances(EAs).rar. As shown in Table 1, the instances are indexed according to above naming rules and three relationships.

To compare the performance fairly, similar to Algorithm 3, the *S-GROA* is also used in *GA*, *binDE* and ABC_{bin} for handling infeasible solutions of SUKP. Furthermore, for all algorithms, the population size is set to be $N = 20$, and the number of iterations is $MIT = Max\{m, n\}$ for all SUKP instances, where m is the number of items and n is the number of elements in instance.

The other parameters of *GA*, *BABC*, ABC_{bin}, and *binDE* are set as follows.

(1) In GA, the single point cross-over, uniform mutation and roulette wheel selection are employed. The crossover probability is $p_c = 0.8$, and the mutation probability is $p_m = 0.01$.

(2) In *BABC* and ABC_{bin}, we set $a = 5.0$ and $limit = Max\{m, n\}/5$, where m is the number of items and n is the number of elements in SUKP.

(3) In *binDE*, we set the scaling factor $F = 0.5$ and the crossover constant $CR = 0.3$.

5.2. Result analysis and comparison

From Tables 2–4, *Best* denotes the best values of the performance among independent computation by using all algorithms among 100 times. *Mean* and *StD* denote the mean values and the standard deviations among 100 times for all algorithms. *Time* denotes the average time cost of each computation (unit: second). For the approximation algorithm *A-SUKP*, *Best*, *Mean* are the same values (the approximate results), and *Time* denotes the time cost of each execution (unit: second).

According to the results in Table 2, the BABC obtains the largest values of *Best* for 6 instances and the largest values of *Mean* for 8 instances in the first category of SUKP instances. GA achieves 4 largest values of *Best* and 2 largest values of *Mean*. ABC_{bin} and binDE have only one largest value of *Best* respectively. As shown in Table 3, the BABC obtains the largest values of *Best* for 5 instances and the largest values of *Mean* for 1 instances in the second category of SUKP instances. GA achieves 4 largest values of *Best* and 3 largest values of *Mean*. binDE only has only one largest value of *Best*, while ABC_{bin} does not obtain any largest values. Similarly, as shown in Table 4, the BABC obtains the largest values of *Best* for 8 instances and *Mean* for 8 instances in the third category of SUKP instances. GA achieves 2 largest values of *Best* and of *Mean*. binDE only has one largest value of *Best*, while ABC_{bin} does not obtain any largest values.

According to the performance of different methods on three categories of SUKP instances, the BACA has the best performance,

Table 4
The computing results of the third kind of SUKP instances.

Instance	Results	*A-SUKP*	*GA*	*BABC*	ABC_{bin}	*binDE*
sukp 85_100_0.10_0.75	*Best*	10 231	11 454	**11 664**	11 206	11 352
	Mean	10 231	11 092.7	**11 182.7**	10 879.5	11 075
	StD	0.00	171.22	183.57	163.62	119.42
	Time	0.156	0.113	0.188	0.196	0.193
sukp 85_100_0.15_0.85	*Best*	10 483	12 124	**12 369**	12 006	**12 369**
	Mean	10 483	11 326.3	**12 081.6**	11 485.3	11 875.9
	StD	0.00	417.00	193.79	248.33	336.94
	Time	0.172	0.131	0.217	0.216	0.217
sukp 185_200_0.10_0.75	*Best*	11 508	12 841	**13 047**	12 308	13 024
	Mean	11 508	12 236.6	**12 522.8**	11 667.9	12 277.5
	StD	0.00	198.18	201.35	177.14	234.24
	Time	2.766	1.231	1.502	1.588	1.515
sukp 185_200_0.15_0.85	*Best*	8621	**10 920**	10 602	10 376	10 547
	Mean	8621	**10 351.5**	10 150.6	9684.33	10 085.4
	StD	0.00	208.08	152.91	184.84	160.60
	Time	3.141	1.204	1.948	1.710	1.648
sukp 285_300_0.10_0.75	*Best*	9961	10 994	**11 158**	10 269	11 152
	Mean	9961	10 640.1	**10 775.9**	9957.09	10 661.3
	StD	0.00	126.84	116.80	141.48	149.84
	Time	14.329	3.827	5.450	5.500	5.054
sukp 285_300_0.15_0.85	*Best*	9618	**11 093**	10 528	10 051	10 528
	Mean	9618	**10 190.3**	9897.92	9424.15	9832.32
	StD	0.00	249.76	186.53	197.14	232.72
	Time	16.470	3.990	5.571	5.800	5.720
sukp 385_400_0.10_0.75	*Best*	8672	9799	**10 085**	9235	9883
	Mean	8672	9432.82	**9537.5**	8904.94	9314.57
	StD	0.00	163.84	184.62	111.85	191.59
	Time	45.815	9.325	13.012	12.314	13.149
sukp 385_400_0.15_0.85	*Best*	8064	9173	**9456**	8932	9352
	Mean	8064	8703.66	**9090.03**	8407.06	8846.99
	StD	0.00	154.15	156.69	148.52	210.91
	Time	53.972	9.911	13.724	13.648	14.245
sukp 485_500_0.10_0.75_	*Best*	9559	10 311	**10 823**	10 357	10 728
	Mean	9559	9993.16	**10 483.4**	9615.37	10 159.4
	StD	0.00	117.73	228.34	151.41	198.49
	Time	114.412	18.708	27.227	24.540	23.153
sukp 485_500_0.15_0.85_	*Best*	8157	9329	**9333**	8799	9218
	Mean	8157	8849.46	**9085.57**	8347.82	8919.64
	StD	0.00	141.84	115.62	122.65	168.90
	Time	133.226	20.129	28.493	27.570	26.010

while GA, binDE, and ABC_{bin} has the second, third and worst performance respectively.

In the following, we employ the histogram and line charts to compare the average solving speeds and the average solving results of *BABC*, *GA*, ABC_{bin}, *binDE*, and *A-SUKP*; and then use the histogram to compare the robustness of *BABC*, *GA*, ABC_{bin}, and *binDE*.

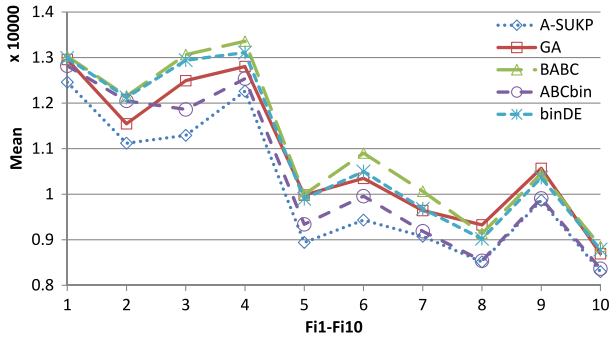
From Fig. 1(a),(b) and (c), by comparing the average time cost of *GA*, *BABC*, ABC_{bin} , *binDE* and *A-SUKP*, we observe that the solving speed of *GA*, *BABC*, ABC_{bin} and *binDE* will become faster than *A-SUKP* with the growth of SUKP instance size $Max\ \{m, n\}$; the solving speed of *GA*, *BABC*, ABC_{bin} and *binDE* is faster four times or above than *A-SUKP* when $Max\{m, n\} \geq 500$. Therefore, from the perspective of solving speed, it is clear that *GA*, *BABC*, ABC_{bin} and *binDE* are faster than *A-SUKP*. They are more suitable for solving SUKP within the limited time. Additionally, the GA has a slightly faster average solving speed than the other three methods.

From the Fig. 2(a), (b) and (c), by comparing the fitting curve of mean values of *GA*, *BABC*, ABC_{bin} and *binDE*, we observe that the computational results of *GA*, *BABC*, ABC_{bin} and *binDE* are better than *A-SUKP* and *BACA* achieves the best performance. Although the mean results of ABC_{bin} in solving SUKP instance Si1, Si7, Si8, Ti5 and Ti6 are slightly worse than *A-SUKP*, it has a better overall performance. Therefore, from the perspective of computational results, we find that *GA*, *BABC*, ABC_{bin} and *binDE* achieves better results than *A-SUKP*. They are more suitable for solving SUKP.

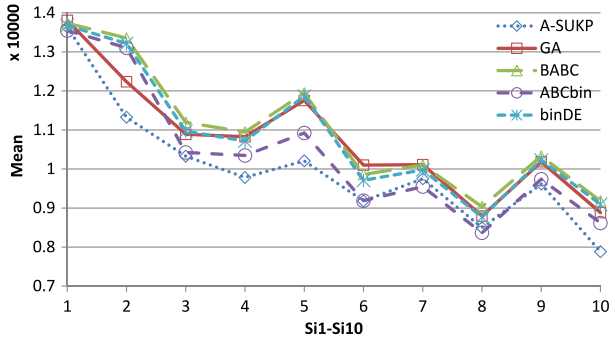
The proposed *S-GROA* is a generic framework which can be employed in different EA algorithms such as GA, BABC, ABC_{bin} and binDE. The performance of adopting *S-GROA* in these three EAs are shown in Fig. 3(a) to (c) in terms of solving three kinds of SUKP instances. To illustrate the applicability of the framework, the classical EAs such as GA and binDE are firstly employed. ABCbin, a quite classical in terms of solving binary optimization problems, is also adopted. From Fig. 3(a), we observe that the standard deviation *StD* is less than 150 in most cases for *BABC* and *binDE*. We thus conclude that *BABC* and *binDE* are the most robust for the first category of SUKP instances. From Fig. 3(b) and (c), we find that the standard deviation *StD* is less than 200 in most cases for *BABC* and ABC_{bin}. They are the most robust for the second and third categories of SUKP instances. According to above comparison, we draw the following conclusions:

(1) Employing EAs such as *GA*, *BABC*, ABC_{bin} and *binDE* to solve SUKP not only achieves better computational results but also has much faster speed than *A-SUKP*. The results indicates that EAs are more suitable approximate algorithms than *A-SUKP* to solve SUKP.

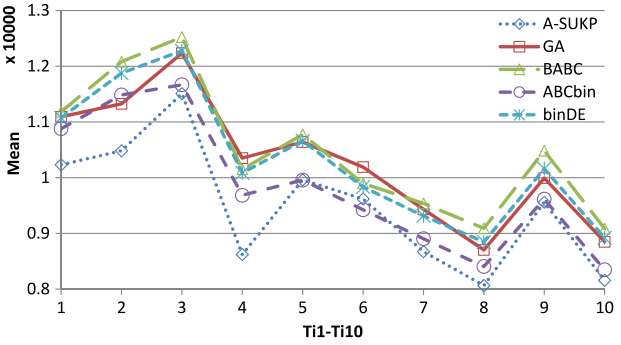
(2) Among all EAs, *BABC* achieves the best computational results and the most robustness with the similar computation speed with other algorithms. The results indicate that *BABC* is not feasible but also more effective.



(a) The first kind of SUKP instances.

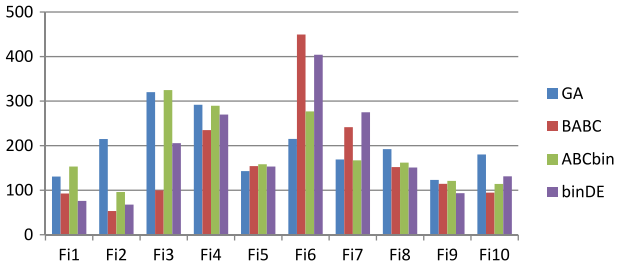


(b) The second kind of SUKP instances.

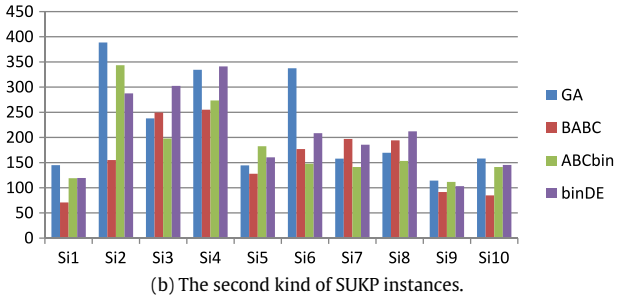


(c) The third kind of SUKP instances.

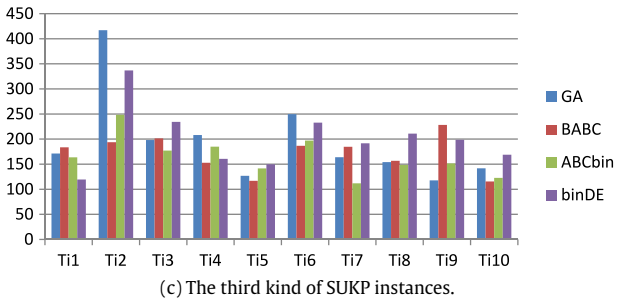
Fig. 2. The *Mean* of *GA*, *BABC*, ABC_{bin}, *binDE* and *A-SUKP* for solving three kinds of SUKP instances.



(a) The first kind of SUKP instances.



(b) The second kind of SUKP instances.



(c) The third kind of SUKP instances.

Fig. 3. The standard deviation of *GA*, *BABC*, ABC_{bin} and *binDE* for solving three kinds of SUKP instances.

6. Conclusion

This paper exploits a full mapping function, which maps one real vector to a 0–1 vector, to propose a novel binary artificial bee colony algorithm *BABC*. To apply *BABC* for solving SUKP, we propose a new approach to solving this problem based on an effective algorithm *S-GROA* which uses the greedy strategy for solving unfeasible solutions. Comparing to approximation algorithms *GA*, ABC_{bin} and *binDE*, *BABC* not only is far more suitable than *A-SUKP* to solve SUKP but also achieves better results than *GA*, ABC_{bin} and *binDE*. In fact, the proposed *S-GROA* is a generic algorithm, which can be adopted in other EAs (e.g., Particle swarm optimization (PSO) [16], Harmony search algorithm (HSA) [36], Artificial algae algorithm (AAA) [37], Fruit fly optimization (FFO) [38], Brain storm optimization (BSO) [39] and Fireworks algorithm (FWA) [40]) for solving SUKP problem to addressing the generated unfeasible solutions.

As SUKP is a strong NPC problem and there is no pseudo-polynomial time-complexity algorithm for solving it, it is worthwhile to find an effective and fast approximate algorithm for this problem. The findings in this article indicate that it is a feasible research direction to design approximation algorithms by using EAs. In our future research, we will continue this direction by exploiting EAs such as PSO, HSA, AAA, FFO, BSO and FWA to solve SUKP, and find the EAs with the best performance. Another two research directions are (i) to consolidate recent machine learning methods [41–43] have a more accurate discriminator; (ii) to apply the proposed method to the multimedia data compression [44–46]; and (iii) to adopt the self-adaptive framework to boost the robustness and accuracy [47].

Acknowledgments

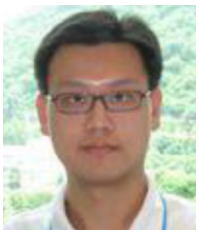
We thank Editor-in-Chief and anonymous reviewers whose valuable comments and suggestions help us significantly improve this article. The first author and corresponding authors contributed equally the same to this article which was supported by Basic Research Project of Knowledge Innovation Program in Shenzhen (JCYJ20150324140036825), China Postdoctoral Science Foundations (2015M572361 and 2016T90799), National Natural Science Foundations of China (61503252 and 71371063), Scientific Research Project Program of Colleges and Universities in Hebei Province (ZD2016005), and Natural Science Foundation of Hebei Province (F2016403055). Haoran Xie's work was supported by the Start-Up Research Grant (RG 37/2016-2017R) and the Internal Research Grant (RG 66/2016–2017) of The Education University of Hong Kong.

References

[1] O. Goldschmidt, D. Nehme, G. Yu, Note: On the set-union knapsack problem, Naval Res. Logist. 41 (6) (1994) 833–842.
[2] Ashwin Arulselvan, A note on the set union knapsack problem, Discrete Appl. Math. 169 (2014) 214–218.
[3] Hans Kellerer, Ulrich Pferschy, David Pisinger, Knapsack Problems, Springer, Berlin, 2004.
[4] S. Khuller, A. Moss, J. Naor, The budgeted maximum coverage problem, Inform. Process. Lett. 70 (1999) 39–45.
[5] C.S. Tang, E.V. Denardo, Models arising from a flexible manufacturing machine, part 11: Minimizing the number of switching instants, Oper. Res. 36 (1988) 778–784.
[6] O. Goldschmidt, D. Nehme, G. Yu, On a Generalization of the Knapsack Problem with Applications to Flexible Manufacturing Systems and Database Partitioning. Working Paper No. 92193-3-7, Graduate School of Business, University of Texas at Austin, 1992.
[7] S. Navathe, S. Ceri, G. Wiederhold, J. Dou, Vertical partitioning algorithms for database design, ACM Trans. Database Syst. 9 (1984) 680–710.
[8] Manghui Tu, Liangliang Xiao, System resilience enhancement through modularization for large scale cyber systems, in: 2016 IEEE/CIC International Conference on Communications in China, IEEE, 2016.
[9] Xinan Yang, Alexei Vernitski, Laura Carrea, An approximate dynamic programming approach for improving accuracy of lossy data compression by Bloom filters, European J. Oper. Res. 252 (3) (2016) 985–994.
[10] Bruce Schneier, Appiled Cryptography: Protocols, Algorithms, and Source Code in C, second ed., John Wiley & Sons, Inc., New Jersey, 1996.
[11] Dervis Karaboga, An idea based on honey bee swarm for numerical optimization. Technical report. Computer Engineering Department, Engineering Faculty, Erciyes University, 2005.
[12] Yaosheng Liang, Zhongping Wan, Debin Fang, An improved artificial bee colony algorithm for solving constrained optimization problems, Int. J. Mach. Learn. Cyb. 8 (3) (2017) 739–754.
[13] D. Karaboga, B. Akay, A survey: algorithms simulating bee swarm intelligence, Artif. Intell. Rev. 31 (2009) 61–85.
[14] Lothar M. Schmitt, Theory of genetic algorithms, Theoret. Comput. Sci. 259 (1–2) (2001) 1–61.
[15] R. Storn, K. Price, Differential evolution—A simple and efficient heuristic for global optimization over continuous spaces, J. Global Optim. 11 (1997) 341–359.
[16] J. Kennedy, R.C. Eberhart, Particle swarm optimization, in: Proceedings of the IEEE International Conference on Neural Networks (Perth, Australia), Vol. IV, IEEE Service Center, Piscataway, NJ, 1995, pp. 1942–1948.
[17] N. Karaboga, A new design method based on artificial bee colony algorithm for digital iir filters, J. Frankl. Inst.-Eng. Appl. Math. 346 (4) (2009) 328–348.
[18] D. Karaboga, C. Ozturk, Neural networks training by artificial bee colony algorithm on pattern classification, Neural Netw. World 19 (3) (2009) 279–292.
[19] Fei Kang, Junjie Li, Zhenyue Ma, Rosenbrock artificial bee colony algorithm for accurate global optimization of numerical functions, Inform. Sci. 181 (2011) 3508–3531.
[20] D. Karaboga, C. Ozturk, A novel clustering approach: Artificial bee colony (abc) algorithm, Appl. Soft Comput. 11 (1) (2011) 652–657.
[21] Quan-Ke Pan, M. Fatih Tasgetiren, P.N. Suganthan, T.J. Chua, A discrete artificial bee colony algorithm for the lot-streaming flow shop scheduling problem, Inform. Sci. 181 (2011) 2455–2468.
[22] Hsing-Chih Tsai, Integrating the artificial bee colony and bees algorithm to face constrained optimization problems, Inform. Sci. 258 (2014) 80–93 81.
[23] Mustafa Servet Kiran, Huseyin Hakli, Mesut Gunduz, Harun Uguz, Artificial bee colony algorithm with variable search strategy for continuous optimization, Inform. Sci. 300 (2015) 140–157.
[24] Akbar Banitalebi, Mohd Ismail Abd Aziz, Arifah Bahar, Zainal Abdul Aziz, Enhanced compact artificial bee colony, Inform. Sci. 298 (2015) 491–511.
[25] Mustafa Servet Kiran, The continuous artificial bee colony algorithm for binary optimization, Appl. Soft Comput. 33 (2015) 15–23.
[26] Celal Ozturk, Emrah Hancer, Dervis Karaboga, A novel binary artificial bee colony algorithm based on genetic operators, Inform. Sci. 297 (2015) 154–170.
[27] Dinu Calin Secui, A new modified artificial bee colony algorithm for the economic dispatch problem, Energy Convers. Manage. 89 (2015) 43–62.
[28] A.P. Engelbrecht, G. Pampara, Binary differential evolution strategies, in: IEEE Congress on Evolutionary Computation, 2007, pp. 1942–1947.
[29] T.P. Runarsson, X. Yao, Stochastic ranking for constrained evolutionary optimization, IEEE Trans. Evol. Comput. 4 (3) (2000) 284–294.
[30] C.A.C. Coello, Theoretial and numerical constraint-handling techniques used with evolutionary algorithm-A survey of the state of art, Comput. Methods Appl. Mech. Engrg. 191 (2002) 1245–1287.
[31] Z. Michalewicz, Genetic Algorithm+Data Structure=Evolution Programs, Springer-Verlag, Berlin, 1996.
[32] He Yi-chao, Liu Kun-qi, Zhang Cui-jun, Zhang Wei, Greedy genetic algorithm for solving knapsack problem, Comput. Eng. Des. 28 (11) (2007) 2655–2657 2681.
[33] Yichao He, Xinlu Zhang, Wenbin Li, et al., Algorithms for randomized time-varying knapsack problems, J. Comb. Optim. 31 (1) (2016) 95–117.
[34] Yi-Chao He, Xi-Zhao Wang, Yu-Lin He, Shu-Liang Zhao, Wen-Bin Li, Exact and approximate algorithms for discounted {0-1} knapsack problem, Inform. Sci. 369 (2016) 634–647.
[35] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, second ed., The MIT Press, Cambridge, 2001.
[36] M. Mahdavi, M. Fesanghary, E. Damangir, An improved harmony search algorithms for solving optimization problems, Appl. Math. Comput. 188 (2) (2007) 1567–1579.
[37] Sait Ali Uymaz, Gulay Tezel, Esra Yel, Artificial algae algorithm (AAA) for nonlinear global optimization, Appl. Soft Comput. 31 (2015) 153–171.
[38] Wen-Tsao Pan, A new fruit fly optimization algorithm: Taking the financial distress model as an example, Knowl.-Based Syst. 26 (2012) 69–74.
[39] Shi Cheng, Quande Qin, Junfeng Chen, Yuhui Shi, Brain storm optimization algorithm: a review, Artif. Intell. Rev. (2016), http://dx.doi.org/10.1007/s10462-016-9471-0.
[40] Ying Tan, Yuanchun Zhu, Fireworks algorithm for optimization, in: Y. Tan, Y. Shi, K.C. Tan (Eds.), ICSI 2010, Part I, in: LNCS, vol. 6145, Springer-Verlag, Berlin Heidelberg, 2010, pp. 355–364.
[41] B. Gu, V.S. Sheng, Z. Wang, D. Ho, S. Osman, S. Li, Incremental learning for ν-support vector regression, Neural Netw. 67 (2015) 140–150.
[42] Sungwan Bang, Jongkyeong Kang, Hierarchically penalized support vector machine with grouped variables, Int. J. Mach. Learn. Cyb. 8 (4) (2017) 1211–1221.
[43] Magdalene Marinaki, Yannis Marinakis, A bumble bees mating optimization algorithm for the feature selection problem, Int. J. Mach. Learn. Cyb. 7 (4) (2016) 519–538.
[44] Z. Pan, J. Lei, Y. Zhang, X. Sun, S. Kwong, Fast motion estimation based on content property for low-complexity H. 265/HEVC encoder, IEEE Trans. Broadcast. 62 (3) (2016) 675–684.
[45] Z. Pan, Y. Zhang, S. Kwong, Efficient motion and disparity estimation optimization for low complexity multiview video coding, IEEE Trans. Broadcast. 61 (2) (2015) 166–176.
[46] Z. Xia, X. Wang, X. Sun, Q. Liu, N. Xiong, Steganalysis of LSB matching using differences between nonadjacent pixels, Multimedia Tools Appl. 75 (4) (2016) 1947–1962.
[47] Y. Xue, J. Jiang, B. Zhao, T. Ma, A self-adaptive articial bee colony algorithm based on global best for global optimization, Soft Comput. (2017). http://dx.doi.org/10.1007/s00500-017-2547-1.

Yichao He is a Professor at Hebei GEO University, China. His research interest includes combinatorial optimization, algorithm theory and intelligent computing. He has published over 50 publications including Journal of Combinatorial Optimization, Journal of Computational Biology and so on.

Haoran Xie is an Assistant Professor at The Education University of Hong Kong. He received his Ph.D. in Computer Science from the City University of Hong Kong. His research interests include deep learning, big data, financial and educational data mining. He has published over 90 publications including AAAI, WWW, DASFAA, INTSYS, NEUNET and so on. He is served as guest editors in JMLC, NEUCOM, IJDET and WIJ. He has also served as a co-chair/committee member of WI, TALE, CPSCom, GCCCE, U-Media, WISE and ICWL.



Tak-Lam Wong is an Assistant Professor at The Education University of Hong Kong. He received his Ph.D. in Systems Engineering and Engineering Management from The Chinese University of Hong Kong. His research interests include Web mining, data mining, information extraction, machine learning, and knowledge management. He has published over 100 publications including PAMI, TKDE, TOIS, and TOIT. He also has served as a committee member in more than 20 international conferences.

Xizhao Wang is a Professor at Shenzhen University, China. He received his Ph.D. in computer science from Harbin Institute of Technology on September 1998. From 2000 to 2012 Dr. Wang served in Hebei University as a professor and the dean of school of Mathematics and Computer Sciences. From 2013 to now Dr. Wang worked as a professor in Big Data Institute of ShenZhen University since 2013. Prof. Wang's major research interests include uncertainty modeling and machine learning for big data. Prof. Wang has edited 6+ special issues and published 3 monographs, 2 textbooks, and 150+ peer-reviewed research papers. By the Google scholar, the total number of citations is over 3000 and the maximum number of citation for a single paper is over 200. The H-index is 25 up to March 2015. Prof. Wang is on the list of Elsevier 2015 most cited Chinese authors. As a Principle Investigator (PI) or co-PI, Prof. Wang's has completed 30+ research projects. Prof. Wang is an IEEE Fellow, the previous BoG member of IEEE SMC society, the chair of IEEE SMC Technical Committee on Computational Intelligence, and the Chief Editor of Machine Learning and Cybernetics Journal.